Data Structure and Analysis of Algorithms

CST225-3

Group No: 11

Group No: 11

Individual Contribution

Registration Number	Individual contribution
1. UWU/CST/21/034 - R.K.C. Priyadarshana	Implementation, Advantages and Disadvantages
2. UWU/CST/21/088 - T. Viyaasan	Implementation, Application
3. UWU/CST/21/044 - P.S.H. Gunasekara	Implementation, Introduction
4. UWU/CST/21/030 - H.A.I. Kaushalya	Implementation, Presentation
5. UWU/CST/21/047 - A.M.T.S. Adikari	Implementation, Presentation
6. UWU/CST/21/060 - C.C.R. Jayawardana	Implementation, Algorithm Description, Conclusion
7. UWU/CST/21/038 - I.C Dewamiththa	Implementation, Algorithm Description, Conclusion

Table of Contents

1. Introduction

Dijkstra's Algorithm, named after the Dutch computer scientist Edsger W. Dijkstra who proposed it in 1956, is a fundamental algorithm in computer science and graph theory. It is designed to solve the shortest path problem for a graph with non-negative edge weights, providing an efficient method for finding the shortest path from a single source vertex to all other vertices in the graph. This algorithm is widely used in various applications such as network routing, geographical mapping, and even in video games for pathfinding.

The algorithm operates on a weighted graph, which consists of a set of vertices connected by edges, each with an associated non-negative weight. By iteratively selecting the vertex with the minimum tentative distance, updating the distances to its neighboring vertices, and marking it as visited, Dijkstra's Algorithm efficiently builds the shortest path tree.

One of the key advantages of Dijkstra's Algorithm is its simplicity and the guarantee of finding the shortest path in a graph with non-negative weights. However, its performance can be impacted by the graph's density and the choice of data structures used for implementation. Understanding Dijkstra's Algorithm provides a foundation for more advanced shortest path algorithms and is essential for solving complex problems in various fields of computer science and operations research.

This report aims to provide a comprehensive overview of Dijkstra's Algorithm, explore its time complexity, discuss its applications, and analyze its strengths and limitations. Through detailed explanations and practical examples, we aim to provide a comprehensive understanding of this essential algorithm.

2. Algorithm Description

Dijkstra's Algorithm is a classical algorithm used to find the shortest path between nodes in a graph, which may represent, for example, road networks. The algorithm is fundamental in the field of graph theory and has numerous practical applications in computer science, especially in network routing and geographic information systems (GIS).

2.1 Basic Principle

Dijkstra's Algorithm operates on a weighted graph, where each edge between nodes has a non-negative weight. The goal is to find the shortest path from a starting node (source) to all other nodes in the graph. It employs a greedy approach, progressively selecting the node with the smallest known distance from the source and exploring its neighbors to find the shortest path.

2.1 Steps of the Algorithm

1. Initialization

- Set the distance to the source node to zero and the distance to all other nodes to infinity.
- Create a priority queue (or a similar data structure) to hold nodes to be explored, initially containing only the source node with a distance of zero.

2. Processing Nodes:

- While the priority queue is not empty, extract the node with the smallest distance (let's call this node 'u').
- For each neighbor of 'u' (node 'v'), calculate the tentative distance from the source to 'v' through 'u'. If this distance is less than the currently known distance to 'v', update the distance to 'v' and add 'v' to the priority queue with the new distance.

3. Updating Distances:

- Continue this process until all nodes have been processed, meaning the shortest path to each node has been found.

2.2 Pseudocode

```
Begin
function Dijkstra(Graph, source):
    dist[source] := 0
    for each vertex v in Graph:
        if v ≠ source:
            dist[v] := infinity
        add v to priority queue Q with priority dist[v]

    while Q is not empty:
        u := vertex in Q with smallest distance in dist[]
        remove u from Q

        for each neighbor v of u:
            alt := dist[u] + length(u, v)
```

```
          if alt < dist[v]:
              dist[v] := alt
              decrease priority of v in Q to alt

    return dist
End
```

2.3 Example

Consider a simple graph with nodes A, B, C, D, and E, and the following weighted edges:

A to B (4)
A to C (2)
B to C (5)
B to D (10)
C to E (3)
D to E (4)

Starting from node A, Dijkstra's Algorithm will find the shortest path to each node as follows:

Initialize distances: A=0, B=∞, C=∞, D=∞, E=∞
Process node A: Update B=4, C=2
Process node C: Update E=5
Process node B: No updates
Process node E: Update D=9
Process node D: No updates

The shortest paths from A are:

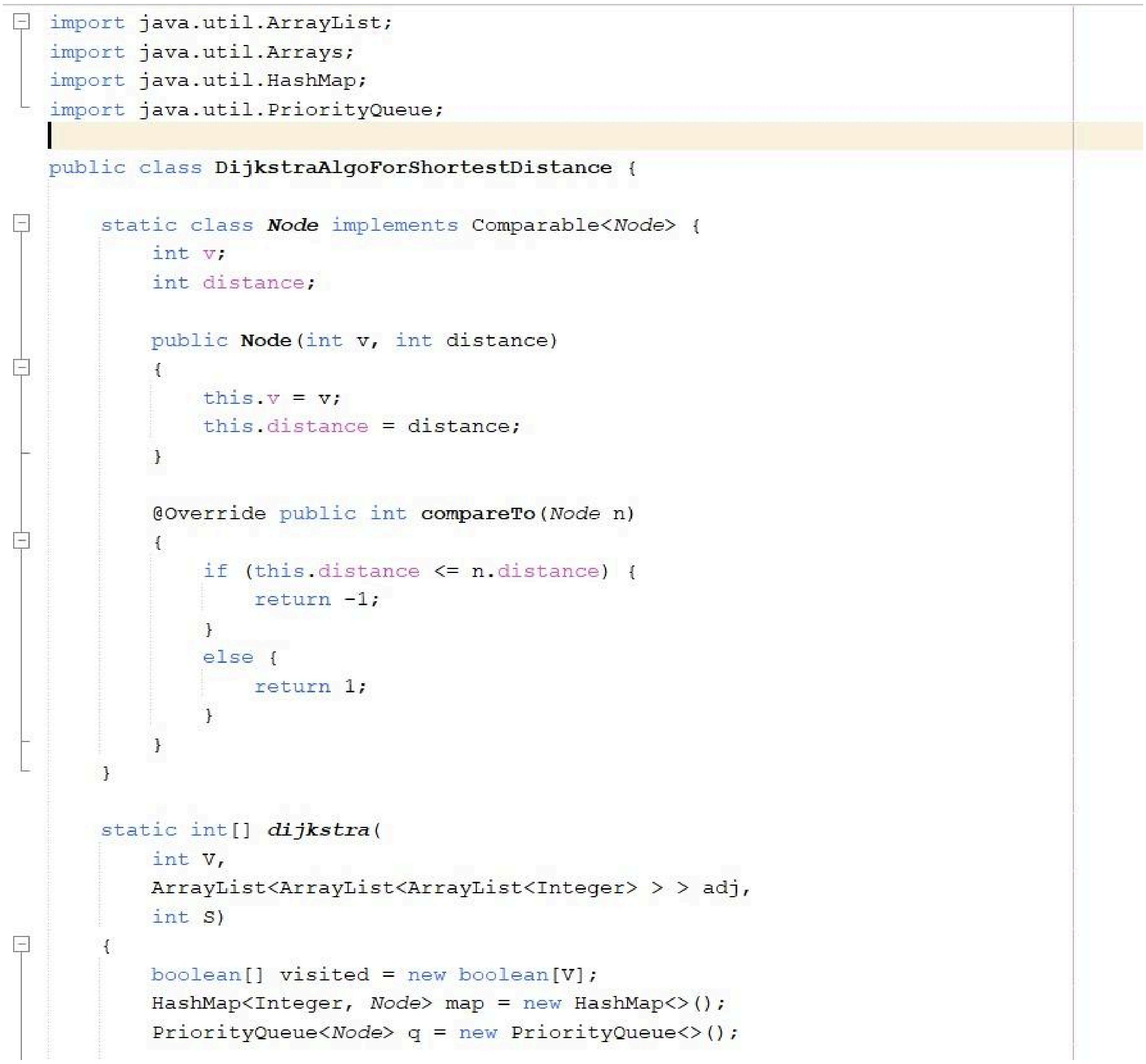
A to B: 4
A to C: 2
A to D: 9
A to E: 5

2.4 Complexity

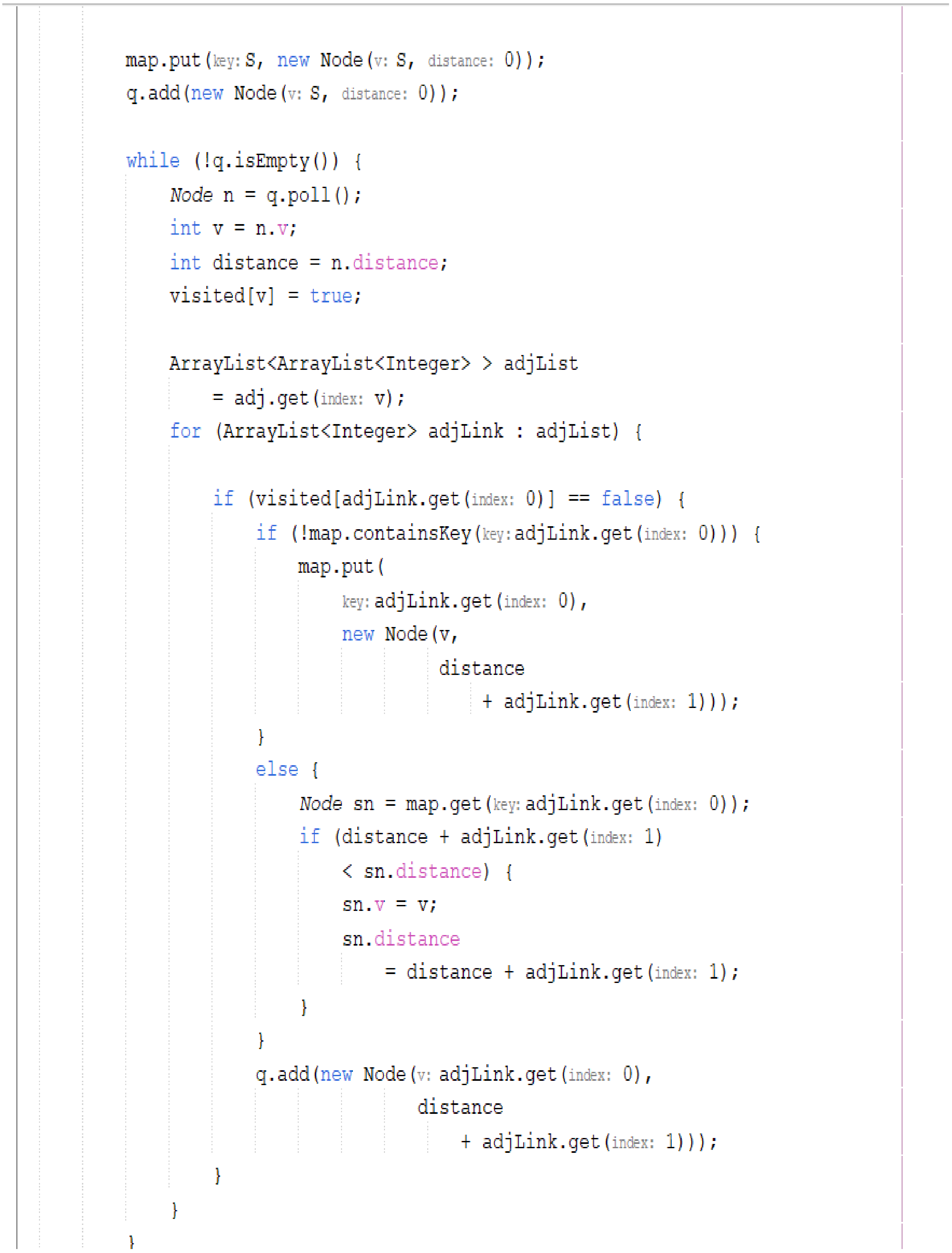
The time complexity of Dijkstra's Algorithm depends on the data structure used for the priority queue. Using a binary heap, the algorithm runs in $O((V + E) \log V)$ time, where V is the number of vertices and E is the number of edges. With a Fibonacci heap, it can be improved to $O(E + V \log V)$.

Dijkstra's Algorithm remains one of the most efficient and widely used algorithms for finding shortest paths in graphs, highlighting its significance in both theoretical and applied computer science.

3. Implementation

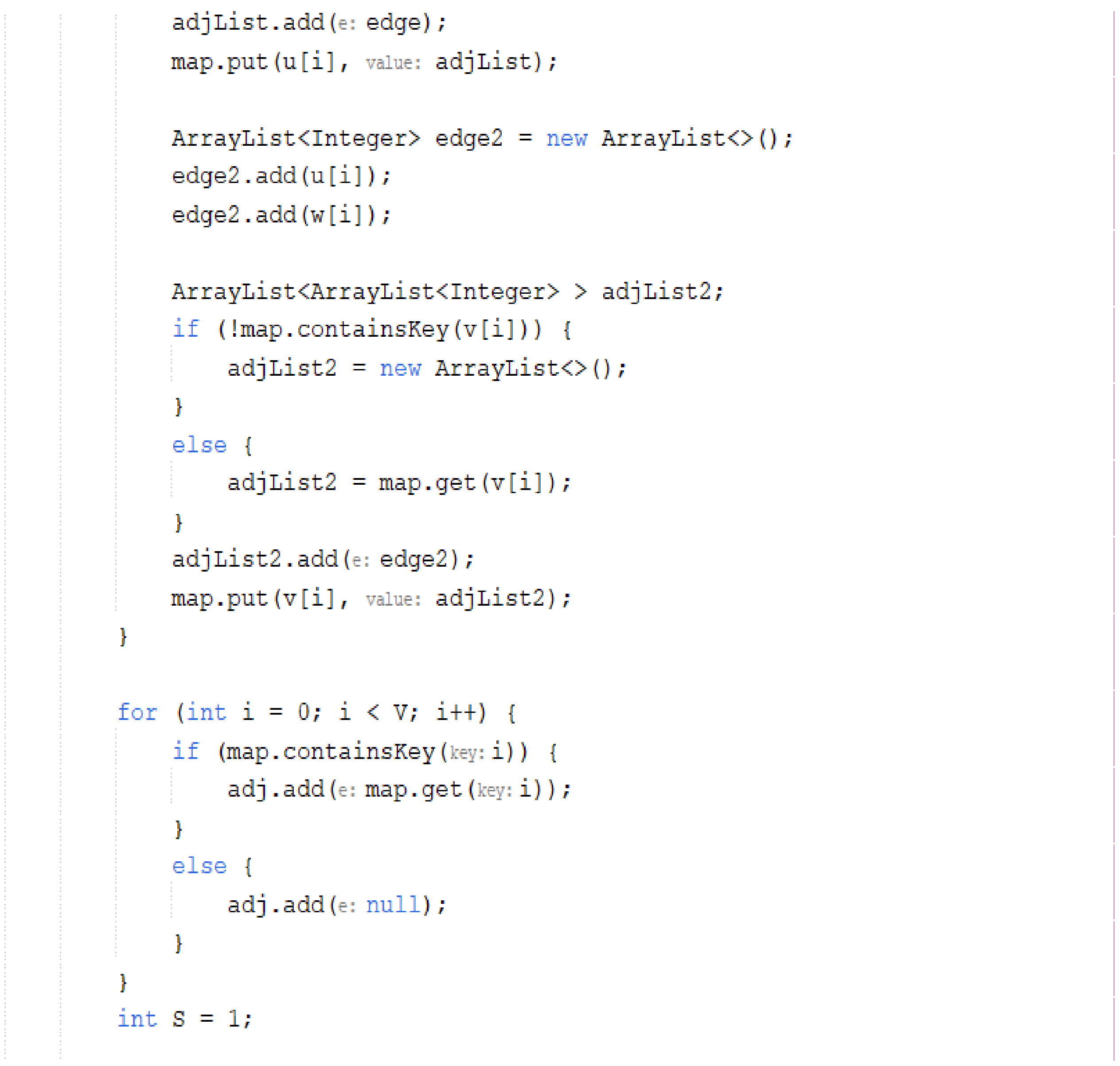


```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.PriorityQueue;

public class DijkstraAlgoForShortestDistance {

    static class Node implements Comparable<Node> {
        int v;
        int distance;

        public Node(int v, int distance)
        {
            this.v = v;
            this.distance = distance;
        }

        @Override public int compareTo(Node n)
        {
            if (this.distance <= n.distance) {
                return -1;
            }
            else {
                return 1;
            }
        }
    }

    static int[] dijkstra(
        int V,
        ArrayList<ArrayList<ArrayList<Integer> > > adj,
        int S)
    {
        boolean[] visited = new boolean[V];
        HashMap<Integer, Node> map = new HashMap<>();
        PriorityQueue<Node> q = new PriorityQueue<>();
```

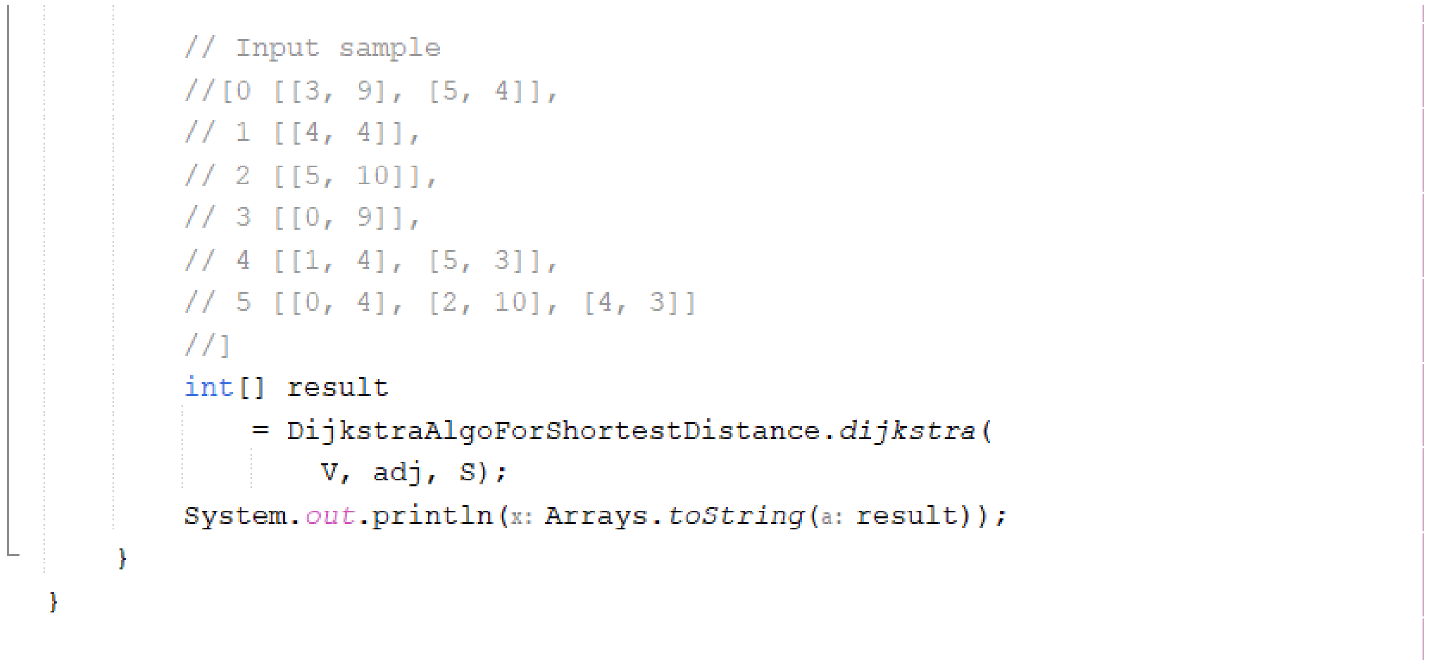


```java
map.put(key: S, new Node(v: S, distance: 0));
q.add(new Node(v: S, distance: 0));

while (!q.isEmpty()) {
    Node n = q.poll();
    int v = n.v;
    int distance = n.distance;
    visited[v] = true;

    ArrayList<ArrayList<Integer> > adjList
        = adj.get(index: v);
    for (ArrayList<Integer> adjLink : adjList) {

        if (visited[adjLink.get(index: 0)] == false) {
            if (!map.containsKey(key: adjLink.get(index: 0))) {
                map.put(
                    key: adjLink.get(index: 0),
                    new Node(v,
                             distance
                                 + adjLink.get(index: 1)));
            }
            else {
                Node sn = map.get(key: adjLink.get(index: 0));
                if (distance + adjLink.get(index: 1)
                    < sn.distance) {
                    sn.v = v;
                    sn.distance
                        = distance + adjLink.get(index: 1);
                }
            }
            q.add(new Node(v: adjLink.get(index: 0),
                           distance
                               + adjLink.get(index: 1)));
        }
    }
}
```

```java
        int[] result = new int[V];
        for (int i = 0; i < V; i++) {
            result[i] = map.get(key: i).distance;
        }

        return result;
    }

    public static void main(String[] args)
    {
        ArrayList<ArrayList<ArrayList<Integer> > > adj
            = new ArrayList<>();
        HashMap<Integer, ArrayList<ArrayList<Integer> > >
            map = new HashMap<>();

        int V = 6;
        int E = 5;
        int[] u = { 0, 0, 1, 2, 4 };
        int[] v = { 3, 5, 4, 5, 5 };
        int[] w = { 9, 4, 4, 10, 3 };

        for (int i = 0; i < E; i++) {
            ArrayList<Integer> edge = new ArrayList<>();
            edge.add(v[i]);
            edge.add(w[i]);

            ArrayList<ArrayList<Integer> > adjList;
            if (!map.containsKey(u[i])) {
                adjList = new ArrayList<>();
            }
            else {
                adjList = map.get(u[i]);
            }
```



```
            adjList.add(e: edge);
            map.put(u[i], value: adjList);

            ArrayList<Integer> edge2 = new ArrayList<>();
            edge2.add(u[i]);
            edge2.add(w[i]);

            ArrayList<ArrayList<Integer> > adjList2;
            if (!map.containsKey(v[i])) {
                adjList2 = new ArrayList<>();
            }
            else {
                adjList2 = map.get(v[i]);
            }
            adjList2.add(e: edge2);
            map.put(v[i], value: adjList2);
        }

        for (int i = 0; i < V; i++) {
            if (map.containsKey(key: i)) {
                adj.add(e: map.get(key: i));
            }
            else {
                adj.add(e: null);
            }
        }
        int S = 1;
```



```
        // Input sample
        //[0 [[3, 9], [5, 4]],
        // 1 [[4, 4]],
        // 2 [[5, 10]],
        // 3 [[0, 9]],
        // 4 [[1, 4], [5, 3]],
        // 5 [[0, 4], [2, 10], [4, 3]]
        //]
        int[] result
            = DijkstraAlgoForShortestDistance.dijkstra(
                V, adj, S);
        System.out.println(x: Arrays.toString(a: result));
    }
}
```

4. Applications

Dijkstra's algorithm is widely used in various real-world applications that require finding the shortest path in a graph. Some of the common applications are shown below.

1. Network Routing

- Internet Routing Protocols: Dijkstra's algorithm is used in routing protocols like OSPF (Open Shortest Path First) to find the shortest path for data packets to travel across a network.
- Telecommunications Networks: Optimizing data transmission paths to reduce latency and improve efficiency.

2. GPS Navigation Systems

- Pathfinding: Used in GPS systems to find the shortest or fastest route from a starting location to a destination.
- Traffic Management: Real-time traffic data is used to find alternative routes to avoid congestion.

3. Robotics

- Path Planning: Helps robots navigate from one point to another while avoiding obstacles.
- Autonomous Vehicles: Used in self-driving cars for route planning and obstacle avoidance.

4. Geographic Information Systems (GIS)

- Route Optimization: Finding the best route for transportation, logistics, and delivery services.
- Urban Planning: Analyzing and planning optimal routes for road networks and public transportation systems.

5. Computer Games

- AI Pathfinding: Used in game development to determine the movement of characters and units in a game world.
- Real-Time Strategy Games: Optimizing unit movements and finding paths in dynamic game environments.

6. Operations Research

- Logistics and Supply Chain Management: Optimizing the movement of goods and resources within a supply chain.
- Facility Layout Planning: Designing optimal layouts for facilities to minimize transportation costs.

7. Social Networks

- Recommendation Systems: Finding the shortest path or degree of separation between users to suggest friends or connections.
- Influence Propagation: Analyzing how information spreads through a network.

8. Healthcare

- Medical Imaging: Analyzing and processing images to find optimal paths or connections.
- Resource Allocation: Optimizing the allocation and routing of medical resources and emergency services.
    -

9. Energy Distribution

- Smart Grids: Optimizing the distribution of electricity in a smart grid to reduce losses and improve efficiency.
- Pipeline Networks: Finding optimal paths for oil, gas, and water pipelines.

10. Public Transportation Systems

- Route Planning: Designing efficient public transportation routes and schedules.
- Real-Time Navigation: Providing real-time updates and alternative routes based on current conditions.

5. Advantages and Disadvantages

Advantages and disadvantages of the Dijkstra's Algorithm

Advantages	Disadvantages

Dijkstra's algorithm can be very efficient with a time complexity of O(V2)O(V^2)O(V2) for a graph with VVV vertices when using a simple array, or O((V+E)logV)O((V + E) \log V)O((V+E)logV) with a priority queue (using a Fibonacci heap).	The algorithm does not work with graphs containing negative weight edges. For such graphs, the Bellman-Ford algorithm is more suitable.
It guarantees finding the shortest path from the source node to all other nodes in the graph when edge weights are non-negative.	It requires maintaining a list of the shortest path estimates and a set of visited nodes, which can consume a significant amount of memory for large graphs.
The algorithm is relatively easy to understand and implement compared to other shortest path algorithms like Bellman-Ford.	Although Dijkstra's algorithm can be efficient, for very sparse graphs, other algorithms like A* (with appropriate heuristics) can be more performant.
It can be used for both directed and undirected graphs and can handle graphs with various structures and sizes.	For graphs with a very high number of nodes and edges, the computational complexity can still be relatively high, impacting performance.
Widely used in network routing protocols, mapping applications (like GPS), and various optimization problems.	It finds the shortest path from a single source to all other nodes. If shortest paths from multiple sources are required, the algorithm must be run separately for each source, increasing computation time.
It finds the shortest path from the source to all other nodes, which is useful for problems requiring multiple destination shortest paths.	

6. Conclusion

The Dijkstra Algorithm is a strong and basic tool in graph theory and computer science. Its ability to discover the shortest path in graphs with non-negative edge weights makes it useful for a wide range of applications, including network routing and video game creation. While it has limitations, such as performance concerns with big graphs and limits on non-negative weights, its simplicity and effectiveness have cemented its status as a cornerstone method. Understanding Dijkstra's Algorithm not only helps you solve shortest path issues, but it also lays the framework for learning more sophisticated algorithms and methodologies.

7. References

https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/

https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php

https://www.programiz.com/dsa/dijkstra-algorithm